Comparing MapReduce and *Pipeline* Implementations for Counting Triangles

Edelmira Pasarella*
Computer Science Department
Universitat Politècnica de Catalunya
Barcelona, Spain
edelmira@cs.upc.edu

Maria-Esther Vidal†
Fraunhofer IAIS
Bonn, Germany
Universidad Simón Bolívar
Caracas, Venezuela
vidal@cs.uni-bonn.de

Cristina Zoltan
Computer Science Department
Universitat Politècnica de Catalunya
Barcelona, Spain
Universidad Internacional de Ciencia y Tecnología
Panamá
zoltan@cs.upc.edu

A common method to define a parallel solution for a computational problem consists in finding a way to use the *Divide & Conquer* paradigm in order to have processors acting on its own data and scheduled in a parallel fashion. MapReduce is a programming model that follows this paradigm, and allows for the definition of efficient solutions by both *decomposing* a problem into steps on subsets of the input data and *combining* the results of each step to produce final results. Albeit used for the implementation of a wide variety of computational problems, MapReduce performance can be negatively affected whenever the *replication factor* grows or the *size of the input* is larger than the resources available at each processor. In this paper we show an alternative approach to implement the *Divide & Conquer* paradigm, named *dynamic pipeline*. The main features of *dynamic pipelines* are illustrated on a parallel implementation of the well-known problem of counting triangles in a graph. This problem is especially interesting either when the input graph does not fit in memory or is dynamically generated. To evaluate the properties of *pipeline*, a dynamic *pipeline* of processes and an *ad-hoc* version of MapReduce are implemented in the language Go, exploiting its ability to deal with channels and spawned processes. An empirical evaluation is conducted on graphs of different topologies, sizes, and densities. Observed results suggest that *dynamic pipelines* allows for an efficient implementation of the problem of counting triangles in a graph, particularly, in dense and large graphs, drastically reducing the execution time with respect to the MapReduce implementation.

1 Introduction

The *Divide & Conquer* paradigm [3] is an algorithm design schema that enables to solve large and complex computational problems in three stages: *i*) *Divide*: an instance of the problem is partitioned into subproblems; *ii*) *Conquer*: the subproblems are solved independently; *iii*) *Combine*: the solutions of the subproblems are combined to produce the final results. The *Divide & Conquer* paradigm is well-known for giving good complexity results. MapReduce [18] is an implementation schema/programming

*This research is supported in part by funds from the Spanish Ministry for Economy and Competitiveness (MINECO) and the European Union (FEDER funds) under grant COMMAS (ref. TIN2013-46181-C2-1-R)

†This research is partially supported by the European Union Horizon 2020 programme for the project BigDataEurope (GA 644564).

A. Villanueva (Ed.): PROLE 2016
EPTCS 237, 2017, pp. 20–33, doi:10.4204/EPTCS.237.2

paradigm of the *Divide & Conquer* paradigm, extensively used in the implementation of complex problems. The *divide* stage is done by establishing an *equivalence relation* on the set of values which are the images of an input set transformed by a *mapping process*, such that in the *Conquer* stage *reducers* can act on disjoint sets, i.e., each *reducer* acts on a different equivalence class. Finally, other processes collect the *partial results* produced by the reducers to generate the solution in the *combine* stage.

Frameworks that implement the MapReduce schema have had great success and are mostly addressed to run on distributed architectures. Parallelism is a mean for speeding up solutions for computational programs with large amounts of data in memory and that have, in general, a regular behavior. The MapReduce scheme utilizes the Valiant's Bulk Synchronous Parallel (BSP) model of computation [24], and it is defined in terms of a pipe of three stages: *Map*, *Shuffle*, and *Reduce*[1]. Map transforms a domain, where the *equivalence relation* can be established. Shuffle divides the collection into *subcollections* where the reducers can act independently; the communication between processes in the pipe is done via distributed files which act as shared memory for the processors. Some problems require the composition of several MapReduce processes. The number of composed processes is called the *number of passes* the solution requires. MapReduce implementations require that users provide at least the code for the *Map* and for *Reduce* processes, as well as determine the *number of processors* assigned to the solution. Hadoop [25] is a framework that provides a programing file system and operating system abstractions for distributing data and processing. It also enables the evaluation and testing of MapReduce implementations, and can recover itself from system failures. The success of the MapReduce schema for solving problems having massive input data has been extensively reported in the literature [17], however, it is also known the MapReduce approach is not suitable for solving problems that require the existence of a shared global state at execution time and solutions that require several passes.

In this work, we tackle limitations of the MapReduce programing schema, and present an alternative computing approach of the *Divide & Conquer* paradigm for solving problems with massive input data. This implementation is based on a *dynamic pipeline* of processes via an asynchronous model of computation, synchronized by channels. A *dynamic pipeline* is like an ordinary pipeline, but the number of stages is not fixed and are dynamically created at runtime, i.e., *dynamic pipeline* is able to adapt itself to the characteristics of a problem instance.

To be concrete, we consider the problem of triangle counting. This problem is relevant for a wide variety of problems in graph data analytics, query optimization, and graph partitioning, e.g., counting triangles represents a building block for computing the clustering coefficient of a graph. We present an implementation of counting triangles based on two rounds of the MapReduce schema presented by Suri and Vassilvitskii [23], and a *dynamic pipeline* implementation following the approach proposed by Aráoz and Zoltan [2]; features of the Go programming language [11] are used to provide efficient implementations of these approaches.

We empirically evaluate the performance of the *dynamic pipeline* and MapReduce based implementations on a large variety of graphs of different size and density. The observed results suggest that the *dynamic pipelining* implementation outperforms the MapReduce based solution for dense graphs and with a large number of edges; savings in execution time can be of up to two orders of magnitude.

In summary in this paper, we make the following contributions:

- A comparison of the MapReduce and *dynamic pipeline* programing schemas in the resolution of the problem of counting triangles.
- Implementations in the Go language of two algorithms that follow the MapReduce and *dynamic pipeline* programing schemas to solve the problem of counting triangles. These algorithms exploit

[1] Some implementations combine the shuffle step with the Map step.

the main properties of Go, i.e., channels and spawned processes, and correspond to implementations of the *dynamic pipeline* and MapReduce programming schemas under the same conditions.

- An empirical evaluation of the MapReduce and *dynamic pipeline* based algorithms to evaluate the performance of both algorithms in a variety of graphs of different density, topology, and size.

This paper is a revised and extended version of a short paper presented at the Alberto Mendelzon Workshop on Foundations of Data Management (AMW2016) [20]. The work is organized as follows: Section 2 describes the problem of counting triangles, and the main features of the Go programing language, while Section 3 presents the implementations of the problem of triangle counting in both MapReduce and *dynamic pipeline* using the Go language. In Section 4, results of the experimental study are reported and discussed. Finally, we present the concluding remarks and future work in Section 5.

2 Preliminaries

2.1 The Problem of Counting Triangles

The problem of counting triangles in a graph has a simple formulation: Count the number of distinct set of 3 edges taken from a given graph, such that $\{(a,b),(b,c),(c,a)\}$, i.e., the number of complete subgraphs of three nodes of the given graph [1]. Counting triangles is a *building block* for determining the connectivity of a community around a node, representing a relevant problem in the context of network analysis. Specifically, given the size of existing networks, efficiency needs to be ensured, and existing approaches exploit the benefits of parallel computation in MapReduce [7, 19, 23], while others implement approximate solutions to the problem [5, 15, 21]. Parallel MapReduce approaches follow the MapReduce programming schema for efficiently counting triangles; however, intrinsic limitations of the MapReduce programming schema may prevent these approaches from scaling up to large and dense graphs. On the other hand, approximation algorithms rely on estimators of the numbers of edges in a graph to approximate the number of triangles. Nevertheless, as shown by Bar-Yossef et al. [6], theoretical bounds suggest that is impossible to precisely approximate these numbers in general graphs efficiently. Recently, Hu et. al [13, 14] propose efficient algorithms that rely on specific graph representations, e.g., adjacent lists. Albeit effective, these algorithms do not follow the MapReduce programming schema, and they will require a pre-processing phase to generate internal representations of a graph.

We tackle an *exact solution* to the problem, and present two algorithms that exploit the properties of MapReduce and pipeline in the Go programing language. To be concrete, we present Go implementations for two algorithms: a) the one proposed by Suri and Vassilvitskii [23]; b) the algorithm proposed by Aráoz and Zoltan [2] where a graph is represented as a sequence of unordered edges. The algorithm by Aráoz and Zoltan can be naturally extended to a triangle listing algorithm. As a precondition, the problem of counting triangles receives undirected simple graphs, i.e., no multiple edges are admitted; to ensure this requirement multiple edges are filtered in a pre-processing stage.

2.2 Main features of the Go programming language: Channels and Goroutines

Go [11] is a programming language that facilitates efficient implementations of parallel programs, and naturally supports concurrency, as well as processes for automatic memory management and garbage collection. Additionally, Go makes available *goroutines* which are lightweight threads managed by Go during runtime. Goroutines are needed not only for dynamically spawning processes, but for describing processes that resume their work (retaining all the values) when stop being blocked.

Go also provides a mechanism of channels to communicate concurrent goroutines, and pass values from senders to receivers. Receivers always block until there is data to receive. If the channel is unbuffered, the sender blocks until the receiver has received the value. If a channel has a buffer, the sender blocks only until the value has been copied to the buffer; while a buffer is full, this means waiting until some receiver has retrieved a value. The Go model of computation using channels, is a synchronous message passing. Task parallelism is obtained by the rule that all unblocked processes can run in parallel.

These features make Go a suitable programing language for the problem of counting triangles, and enable efficient implementations of both *dynamic pipeline* and MapReduce approaches. Figures 2 and 6 illustrate the structure of the two-round MapReduce solution and the *dynamic pipeline* approach for the studied problem, respectively. Both implementations are explained in next section.

3 Solutions to the Problem of Counting Triangles

Suri and Vassilvitskii [23] present a composition of two MapReduce algorithms for solving the triangle counting problem. In the first MapReduce application, the input is a set of edges of an undirected graph, and the output is a set of 2-length paths having a given responsible node. Each 2-length path with its responsible node is represented by a triple (path-triple). The second application of MapReduce receives the path-triples generated by the previous application of MapReduce and the edge-triples, i.e., the edges present in the original input graph with an empty middle element. For each triple, the pair of its end nodes is used as its key. The reducer task identifies if a path-triple and an edge-triple are in the same cluster. If so, the number of triangles is equal to the cluster size minus one; otherwise, the number of triangles in the cluster is zero. Adding the number of triangles in each cluster gives the total number of triangles in the graph. It is common that the number of reducers coincides with the number of available processors. So the behavior is not smooth in the number of processors.

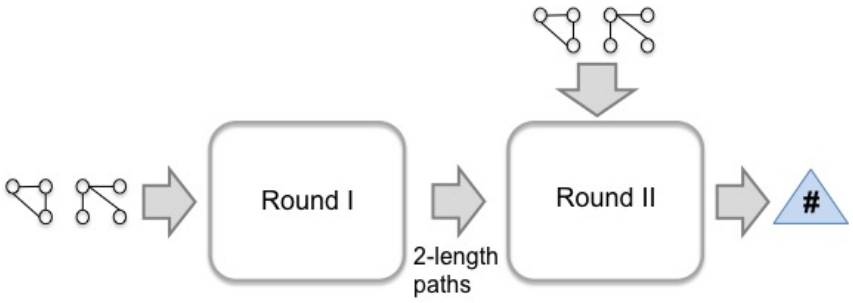


Figure 1: **MapReduce algorithm proposed in [23].** A two-round MapReduce algorithm for counting triangles. Round I generates paths of length 2, while during Round II, edges of the input graph and paths of length 2 are used to count the number of triangles in the input graph

3.1 A MapReduce solution

MapReduce Implementation: Figure 2 shows the phases of our implementation of Suri and Vassilvitskii's algorithm [23]. The program receives as an input a file which is partitioned into as many files as the number of mappers, e.g., the number of available cores. In order to reduce the execution time in the MapReduce implementation, hashing is applied during the Map stage and the mappers communicate via

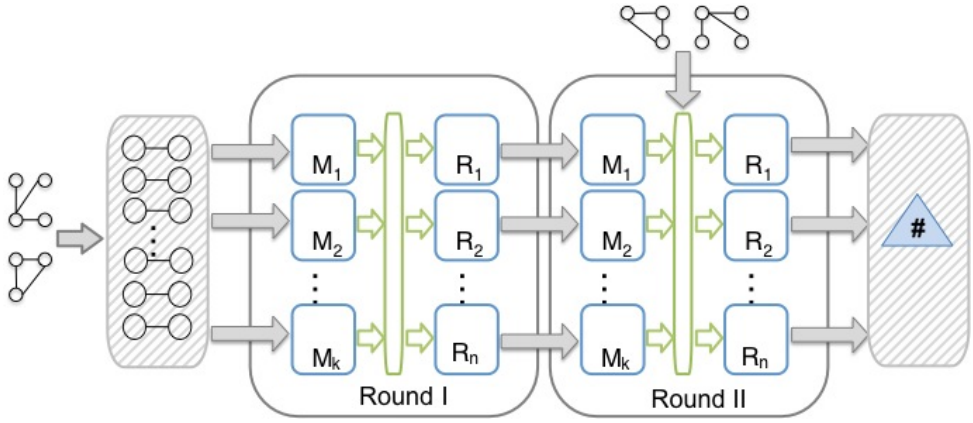


Figure 2: **MapReduce for Triangle Counting**. Go implementation of the two-round of MapReduce algorithm in Figure 1. Small rounded boxes represent mappers and reducers, while large rounded boxes correspond to rounds. Grey and clear arrows represent I/O operations and channel communications, respectively. Mappers and reducers communicate via a channel array. During Round I, edges in the input graph are partitioned to feed the mappers, and reducers generate 2-length paths. In Round II, 2-length paths and edges feed the reducers to count the number of triangles

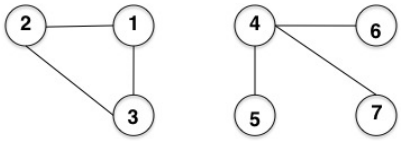
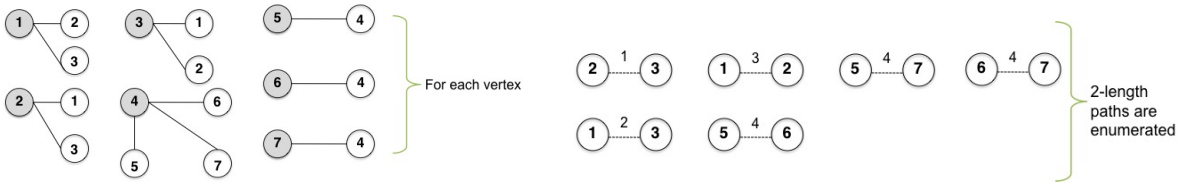


Figure 3: **Running Example.** A graph including only one triangle is used as running example

buffered channels with the reducers. The output of the round I is the set of 2-length paths which are sent to files. In round II, these paths are merged with the input graph edges, and distributed to the reducers.

The output of each reducer is the number of triangles found in its input, i.e., triangles formed by 2-length paths having the same end points and connected by an edge. A process collects the outputs from the reducers to give the final result. Our implementation follows the *MapReduce Online* approach proposed by Condie et. al. [8], and avoids blocking communication between stages.

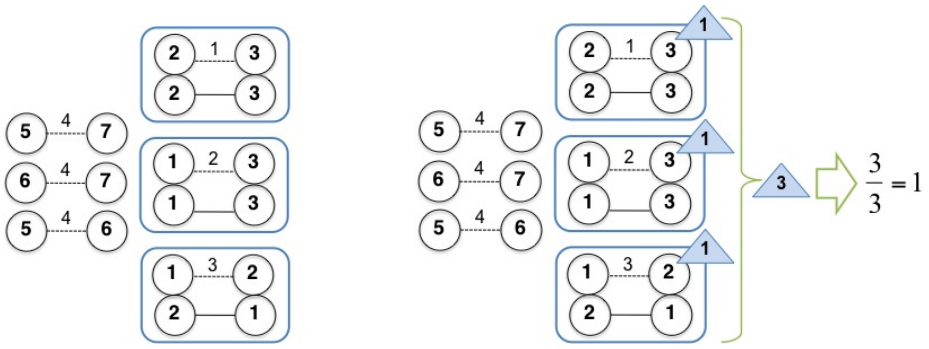


(a) Shuffle Phase: adjacent lists are enumerated

(b) Reduce Phase: 2-length paths are enumerated

Figure 4: **Example of Round I.** Execution of the MapReduce algorithm on the graph of Figure 3. a) Adjacent lists are enumerated during the the Shuffle phase; b) Reducers enumerate paths of length 2

Example 3.1. *Consider the graph shown in Figure 3 where the input to the algorithm is given by the*



(a) Shuffle Phase: Triples are grouped (b) Reduce Phase: triangles are counted

Figure 5: **Example of Round II.** Execution of the MapReduce algorithm on graph in Figure 3. a) Groups of triples are produced during the Shuffle phase; and b) Reducers count triangles in groups of triples

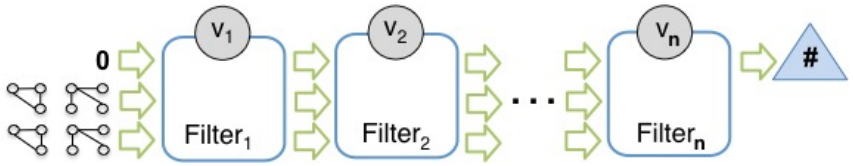


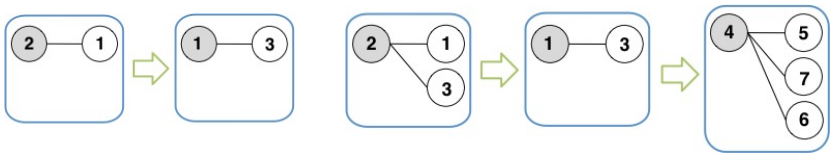
Figure 6: **Pipeline Topology for Triangle Counting**. Dynamic composition of filters that work on set of values not consumed by a previous filter. Rounded boxes represent filters, and grey circles and clear arrows correspond to responsible nodes and data flow, respectively. Filters have three input and three output channels represented by unfilled arrows. The first filter receives the input graph in the second and third channel. The first channel carries the number of triangles; initially, this value is zero. Filters are dynamically created at runtime and executions are adapted to the characteristics of the input

sequence of edges $(2,1),(1,3),(4,5),(2,3),(4,7),(4,6)$. *Figure 4 shows the result at the end of round I, where mappers keep the edges without change, while for each node, the shuffle process produces a set of edges incident to each node i.e., (1,[2,3]), (2,[1,3]), (3,[1,2]), (4,[5,6,7]), (5,[4]), (6,[4]) and (7,[4]) as seen in Figure 4(a). Each reducer produces paths of length 2; middle nodes of these paths are presented as labels in the edges as shown in Figure 4(b). In the running example, 6 paths of length 2 are produced.*

During round II, the mappers transform the edges into triples with an empty middle node. Afterwards, the shuffle groups triples (with a center node or without it) having the same end points as seen in Figure 5(a). A reducer takes the set of values and if the set includes a path of length 2 and one edge, it will give as a result the size of the set minus one; otherwise, the result is 0. We use the notation (a,x,b) to describe a path of length 2 from a to b or an edges from a to b, then Figure 5(b) shows that reducers with input sets that have elements of the form (2,x,3), (1,x,2), and (1,x,3) will output 1 triangle, while the ones having as input (5,x,7), (6,x,7), or (5,x,6) will report 0 triangles. Each triangle is reported 3 times as we can see in Figure 5(b); therefore, the total sum provided by the reducers needs to be divided by 3.

3.2 A Pipeline solution

The pipeline solution is a dynamic composition of filters specialized to nodes of the input graph, and each one works on a set of values not consumed by a previous filter. Each filter has three inputs and three outputs for receiving/sending messages from/to their neighbors. The program structure is a pipe of processes. Initially the pipe is empty, and during execution, the pipe grows/shrinks based on data flowing in the pipe. The first filter is created using the first incoming edge. The first filter will receive the complete set of edges, using the third input. When created, each filter specializes itself with the first incoming edge, using the first node of the edge as *responsible node* and adding the other to an adjacent list. Afterwards, each filter treats the incoming edges, keeping those incident to its responsible node and sending the others to its neighbor. Filters are created dynamically, as new values not consumed by already created filters arrive. As each filter consumes at least one value, no more than $|V|-1$ filters can be created. It is an upper bound because the graph does not have isolated nodes. The number of filters is equal to the number of classes generated by the relations on the original set. The partition relation on edges is created during the execution using responsible nodes as representatives of the set of edges adjacent with the responsible node. The set of responsible nodes is a *dominator set*. Given a graph $G=(V,E)$, a *dominator set* S of G is a subset such that every node $n \in V-S$ is an adjacent node of a node $k \in S$. Whenever there are no more edges in the third input, the filter enters into a second phase. It counts the number of edges flowing in the second input having both endpoints in the adjacency list of the filter responsible node. If there are no more edges flowing in the second input, each filter has the number of triangles, having the responsible node as one of its nodes. Therefore, each triangle is counted only once. Then, the filter enters in the third phase where it outputs the number of triangles already counted and dies; at this point, the pipeline shrinks. The first channel of each filter is used to collect the total number of triangles in the graph. A proof of correctness of this *pipeline* algorithm can be found in [2].

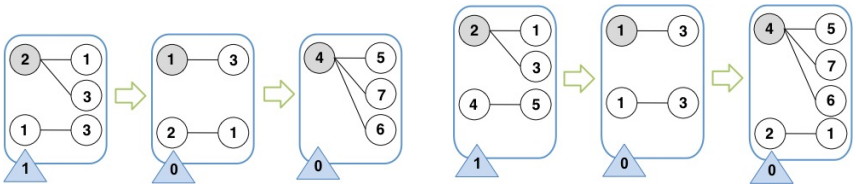


(a) Two filters are created (b) Partition of the edges is generated

Figure 7: **Dynamic Pipeline Partition phase**. Execution of the partitioning phase of the pipeline algorithm. Filters are created according to the input edges. Responsible nodes are represented using grey circles, while filters are modeled using rounded boxes. Arrows represent data flow through the pipeline

The following example illustrates how the algorithm proceeds.

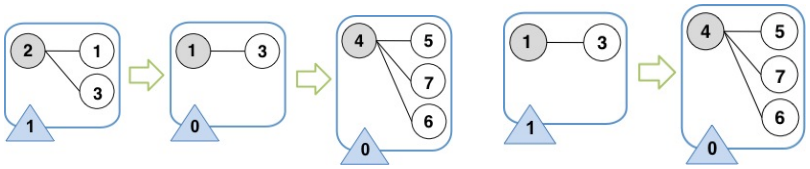
Example 3.2. *Let us consider again the graph shown in Figure 3, where the input to the algorithm is given by the sequence of edges* $(2,1),(1,3),(4,5),(2,3),(4,7),(4,6)$. *Figures 7(a) and 7(b) show the state of the algorithm in the Partition phase. Figure 7(a) presents the state when the input is partially consumed, while Figure 7(b) presents the state after reading and processing all the edges. In Figure 7(a), edges (2,1) and (1,3) are processed and only two filters are created with responsible nodes 2 and 1. At the end of this phase, as shown in Figure 7(b) there are three filters with corresponding responsible nodes: 2, 1, and 4. Further, each filter keeps all the adjacent nodes to the corresponding responsible node. Figure 8 gives snapshots of the state of the algorithm in the Counting phase. At the beginning of this phase, each filter keeps the nodes adjacent to the corresponding responsible node, not consumed by previous*



(a) Edges (2,1) and (1,3) have both been processed by the left-most filter, while the middle filter has processed the edge (2,1)

(b) Edge (4,5), (1,3), and (2,1) have all been processed by the left-most filter; (1,3) and (2,1) have both been processed by the middle filter; while the right-most filter has processed the edge (2,1)

Figure 8: **Pipeline counting triangle execution-Counting phase**. All triangles adjacent to their corresponding responsible node are counted. Responsible nodes are represented using grey circles, while created filters are modeled using rounded boxes. Arrows represent data flow through the dynamic pipeline



(a) Each filter receives the number in triangles in previous filters, adds it to its triangles' account and deactivates itself

(b) The first filter is no longer active and the second one holds the partial count

Figure 9: **Pipeline counting triangle execution-Aggregation phase**. Elasticity of the dynamic pipeline is illustrated. Responsible nodes are represented using grey circles, while created filters are modeled using rounded boxes. Arrows represent data flow through the dynamic pipeline

fiters. The edges flow in the pipe, and in Figure 8(a), we see that $(2,1),(1,3)$ *are being processed, and as a result the filter with responsible node 2 is able to count a triangle (*  *), while the second filter does not count any triangle. In Figure 8(b) the edges* $(2,1),(1,3),(4,5)$ *are processed by the filters, none of them changing the number of triangles having their responsible node as one of its nodes. Figure 9 shows states of phase 3, where the partial count on each filter is transmitted to its neighbor in order to collect the total sum. In Figure 8(b) we can see that after transmitting its triangle count to its neighbour, the first filter dies and the one with responsible node 1, holds the partial triangle count.*

Pipeline Implementation: Figure 6 shows the topology of our implementation of the *pipeline* of the algorithm proposed by Aráoz and Zoltan [2]. In particular, this topology is the composition of filters specialized to nodes of the input graph, and each one works on a set of values that are not consumed by the previous filter. The first filter receives the complete set of edges on the third input. Each new filter specializes itself with the first incoming edge, using the first node of the edge as responsible node and adds the other one to an adjacent list. Afterwards, each filter treats the incoming edges, keeping those nodes belonging to edges incident to its responsible node and sending the others to their neighbor.

Whenever all the edges are being processed, each filter has the nodes of the received edges incident to its responsible node. The number of filters is equal to the number of classes generated by the relation

on the original set. Filters are processes/*goroutines* that communicate via unbuffered channels and each process is specialized by a responsible node. *Goroutines* have three input channels and three output channels. Processes use lists to keep nodes adjacent to the responsible one. In each filter, every incoming edge in the second input is checked if it is incident to two nodes adjacent to the corresponding responsible one. If so, the number of triangles found is increased by one. When there are no more edges, each filter has the total number of triangles in the graph that includes its responsible node. The first channel will carry the number of triangles found by each *goroutines*. A final process adds up the partial results.

4 Experiments

In this section, we present the experimental results of MapReduce and Pipeline implementations for the problem of counting triangles. The goal of the experiment is to analyze the impact of graph properties on time and space complexity of both implementations. We study the following research questions: **RQ1)** Is the Pipeline based implementation able to overcome the *performance* of MapReduce implementation independently of the input graph characteristics?; **RQ2)** Are *density*, *topology*, and *size* of the input graph equally affecting Pipeline and MapReduce implementations?; **RQ3)** Is the *number of cores* equally affecting Pipeline and MapReduce implementations?. The experimental configuration to evaluate these research questions is as follows:

Graph	# Vertices	# Arcs	Density	File size
DSJC.1	1,000	99,258	**0.10**	1.1MB
DSJC.5	1,000	499,652	**0.50**	5.2MB
DSJC.9	1,000	898,898	**0.90**	9.3MB
Fixed-number-arcs-0.1(FNA.1)	10,000	10,000,000	**0.10**	140MB
Fixed-number-arcs-0.5 (FNA.5)	4,472	10,000,000	**0.50**	138MB
Fixed-number-arcs-0.9 (FNA.9)	3,333	10,000,000	**0.90**	136MB
USA-road-d.NY (NY)	264,346	733,846	**1.04E-5**	13MB
Facebook-SNAP(107)	1,911	53,498	**1.47E-2**	0.524MB

Table 1: **Benchmark of Graphs** Graphs of different sizes and densities. Density is defined as $\frac{\#Arcs}{\#Vertices*(\#Vertices-1)}$

Datasets: We compare these two implementations using graphs of different topologies, densities, and sizes.These graphs are part of the 9th DIMACS Implementation Challenge - Shortest Paths[9]; DSJC.1, DSJC.5, and DSJC.9 are graphs with the same number of nodes and different densities, while in Fixed-number-arcs-0.1(FNA.1), Fixed-number-arcs-0.5(FNA.5), and Fixed-number-arcs-0.9(FNA.9), the number of nodes is changed to affect the graph density. USA-road-d.NY and Facebook-SNAP(107)[16] are real-world graphs that correspond to the New York City road network and a Facebook subgraph, respectively. Table 1 describes these graphs in terms of number of vertices, arcs, graph density, and file size.
Metrics: As evaluation metrics, we consider the execution time (ET) and Virtual-memory (VM). ET represents the elapsed time (in seconds) between the submission of a job and completion of the job including the generation of the final results. VM represents the virtual memory consumed by the batch job measured in GB. Both ET and VM are reported by the *qsub* command when a batch job is submitted to the machine [10].

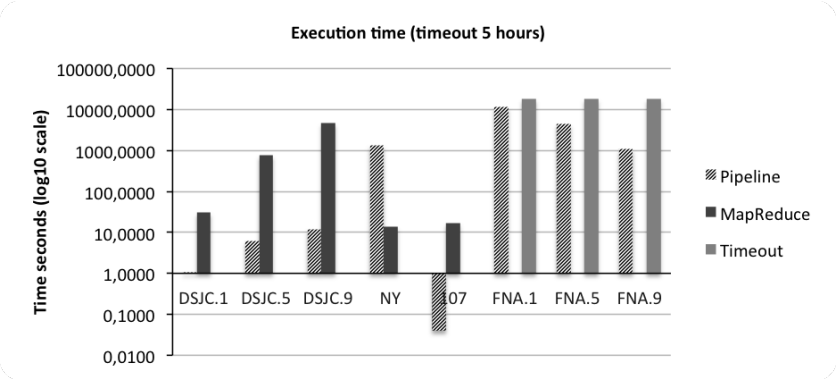


Figure 10: **Execution Time**. For graphs of different size and density, performance of the Pipeline and MapReduce implementations is reported in terms of execution time (ET in log10-scale secs). Jobs that time out at five hours are reported using light grey bars.

Implementation: Programs are run on a node of the cluster of the RDLab-UPC[2] having two processors Intel(R) Xeon(R) CPU X5675 of 3066 MHz with six cores each one. The configuration used in the experiments for submitting jobs is up to 12 cores and 40GB of RAM. Programs are implemented in Go 1.6 [12]. The same job is executed 10 times and the average in reported, given enough shared memory and a timeout of five hours.

Discussion Graphs with different sizes and densities (0.10, 0.50, and 0.90) are evaluated to study our research questions **RQ1** and **RQ2**. Graphs with high density can be considered as the worst case for both program schemes. Plots in Figures 10 and 11 report on execution time (ET in log10-scale secs) and virtual memory (VM in GB) for each of the schemas. Jobs that time out at five hours are reported using grey bars. Jobs for the *pipeline* program in the different graphs are finished in less than 3 hours, while three jobs of the MapReduce implementations do not produce any response in five hours, i.e., these three jobs time out and are reported in light grey bars in Figures 10 and 11. The results suggest that the *pipeline* implementation exhibits the best results in response time and virtual memory consumption for graphs as the ones in DSJC.1, DSJC.5, DSJC.9, FNA.1, FNA.5, and FNA.9. Particularly, in the highly dense graphs, i.e., DSJC.9 and FNA.9, *pipeline drastically reduces* execution time with respect to MapReduce. Similar performance is observed in the *real-world subgraph* of Facebook (Facebook-SNAP(107)), where *pipeline* execution time overcomes MapReduce by three orders of magnitude. Finally, the graph NY that represents the road network of NY city, is *highly sparse* and the *pipeline* implementation generates a large number of processes that the Go scheduler is not able to deal with.

Our benchmark of graphs is also used to evaluate our research question **RQ3**. Plots in Figures 12 and 13 report on execution time (ET secs) for each of the schemas when the number of cores is eight or twelve. Jobs that time out at five hours are reported using light grey bars. For the graphs DSJC.1, DSJC.5, DSJC.9, and 107, jobs of the pipeline implementation requires less than 200 secs. to be completed and produce the response. Similarly, in graphs DSJC.1, (Facebook-SNAP(107)) and NY, jobs of the MapReduce implementations produce the responses in less than 300 secs. As the results reported in Figures 10 and 11, jobs for the MapReduce implementation time out at five hours for large graphs: FNA.1, FNA.5, and FNA.9. This negative performance of MapReduce is caused by the *replication factor* of the problem of counting triangles, i.e., the size of the set of 2-length paths (output in the first phase of MapReduce)

[2] `https://rdlab.cs.upc.edu/`

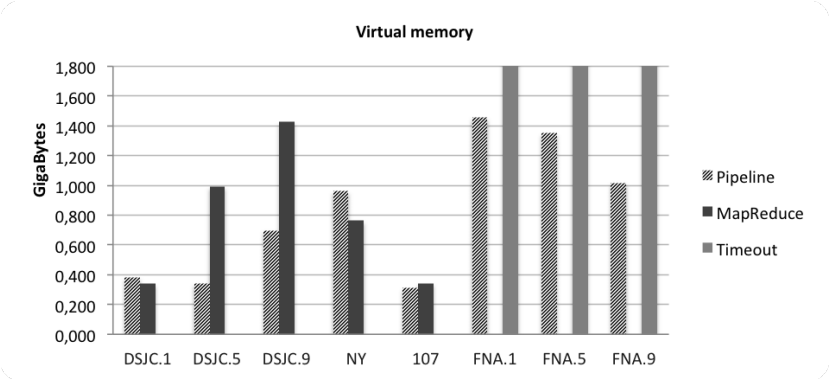


Figure 11: **Virtual Memory**. For graphs of different size and density, performance of the Pipeline and MapReduce implementations is reported in terms of virtual memory (VM in GB). Jobs that time out at five hours are reported using light grey bars.

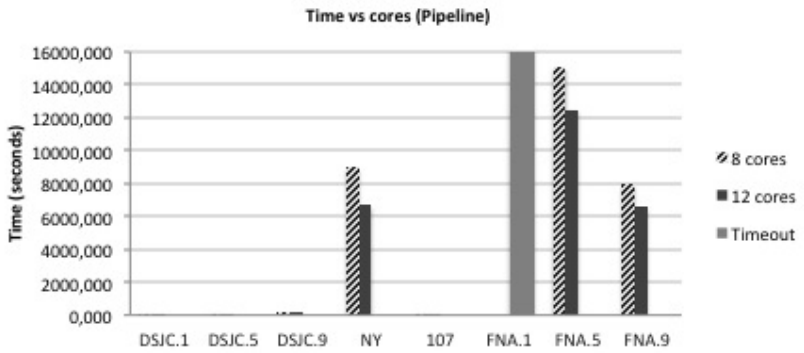


Figure 12: **Impact of the Number of Cores in Pipeline**. For graphs of different size and density, the impact of the number of cores in execution time (ET secs) of the pipeline implementation is reported. For graphs DSJC.1, DSJC.5, DSJC.9, and 107 execution time is less than 200 secs.

is extremely large, O(n^2) where n is the number of graph vertices and these graphs have up to 10,000 vertices. These results corroborate our statement that the *pipeline* programming schema is a *promising* model for implementing complex problem and provides an adaptive solution to the characteristics of the input dataset. Furthermore, *pipeline* is competitive with MapReduce and does not require any previous knowledge of the input dataset.

5 Conclusions and Future Work

We presented an alternative approach, named *dynamic pipeline*, that follows the *Divide & Conquer* paradigm, and relies on a *dynamic pipeline* of processes via an asynchronous model of computation for process communication. Users of the *pipeline* approach need to provide a sequential code or *filters*, and require no understanding of standard concurrency mechanisms, e.g., threads and fine-grain concurrency control, which are aspects known to be difficult to deal with in order to obtain race condition free code in a parallel solution. Contrary to MapReduce, where implementations differ depending on the architec-

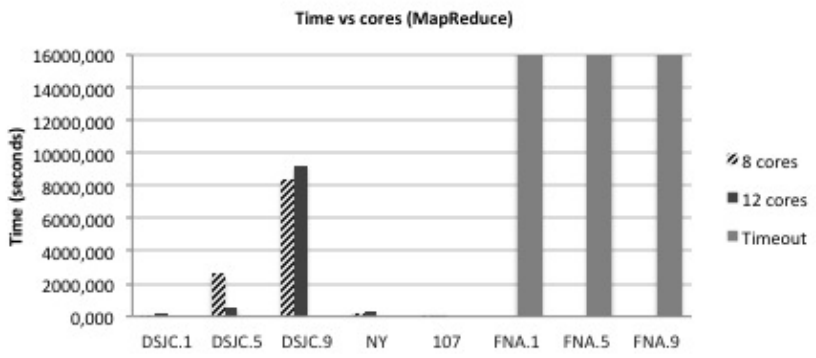


Figure 13: **Impact of Number of Cores in MapReduce**. For graphs of different size and density, impact of the number of cores in execution time (ET secs) of the MapReduce implementation is reported. Jobs that time out at five hours are reported using grey bars. Graphs DSJC.1, (Facebook-SNAP(107)) and NY, jobs of the MapReduce implementations produce the responses in less than 300 secs.

tures, implementations based on the *pipeline* approach can make transparent to users the implementation of channels. The channel abstraction could have several concrete implementations that use shared memory, TCP pipes, or files temporarily persisted in a file system, e.g., as the ones provided by the Dryad distributed technologies [22]. This abstraction can allow for the deployment of the same program in a single machine with several cores, or a net of computing units.

The well-known problem of triangle counting is utilized to illustrate the features of the *pipeline* approach as well as the differences with the MapReduce programming schema. Both programs were implemented in multi-processor nodes. The proposed implementations provide exact solutions for counting triangles by exploiting the main characteristics of the Go programming language, i.e., the evaluation model, a scheduler able to cope with dynamic scheduling, and the notion of channels to enable the communication between processes. The performance of both implementations was empirically evaluated in artificial and real graphs with different sizes and densities. The observed results show a superiority in execution time for the pipeline schema even in dense graphs. The only case where MapReduce exhibits a better performance corresponds to a graph where a large number of nodes have an approximate degree of 2, and this particular configuration results in a program that negatively affects the Go scheduler. The results also suggest that the number of processors has a greater positive impact on the pipeline schema than in MapReduce. Based on these results, we can conclude that the *pipeline* approach is highly scalable, and is able to exhibit performance gains on large problem instances with thousands of tasks, seeming to be most promising when a large number of processors work on shared memory, e.g., in architectures as the one implemented in *The Machine* from Hewlett Packard Labs[3]. In the future, we plan to continue the evaluation of the behavior of the *pipeline* approach in other complex computational problems, and create a programming framework. Further, other algorithms for counting triangles in graph will be implemented and included in our evaluation study, e.g., algorithms by Hu et. al [13, 14]. However, it is important to highlight that because these algorithms require different representations of a graph, e.g., adjacent lists, and are not implemented as MapReduce, they will require a pre-processing phase and will not be able to be used in graphs dynamically generated. In consequence, the experimental evaluation will have to be redefined in order to conduct a fair comparison of the studied approaches.

[3] `http://www.labs.hpe.com/research/themachine/`

Acknowledgements. We would like to thank the reviewers for their insightful comments on the paper; all their comments led us to considerably improve our work. Also, we thank the staff of the *Laboratori de Recerca i Desenvolupament* (RDlab) of the Computer Science Department of the UPC for their support during execution of the experimental evaluation.

References

[1] Noga Alon, Raphael Yuster & Uri Zwick (1997): *Finding and Counting Given Length Cycles*. Algorithmica 17(3), pp. 209–223, doi:10.1007/BF02523189. Available at `http://link.springer.com/article/10.1007/BF02523189`.

[2] Julián Aráoz & Cristina Zoltan (2015): *Parallel Triangles Counting Using Pipelining*. `http://arxiv.org/pdf/1510.03354.pdf`. Available at `http://arxiv.org/abs/1510.03354`.

[3] Mikhail J. Atallah & Marina Blanton, editors (2010): *Algorithms and Theory of Computation Handbook: General Concepts and Techniques*, 2 edition. Chapman & Hall/CRC.

[4] Jost Berthold, Mischa Dieterle & Rita Loogen (2009): *Implementing parallel Google map-reduce in Eden*. In: *Euro-Par 2009 Parallel Processing*, Springer, pp. 990–1002, doi:10.1007/978-3-642-03869-3_91.

[5] Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela & Christian Sohler (2006): *Counting triangles in data streams*. In: *Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 26-28, 2006, Chicago, Illinois, USA*, pp. 253–262, doi:10.1145/1142351.1142388. Available at `http://dl.acm.org/citation.cfm?doid=1142351.1142388`.

[6] Shumo Chu & James Cheng (2012): *Triangle Listing in Massive Networks*. *ACM Trans. Knowl. Discov. Data* 6(4), pp. 17:1–17:32, doi:10.1145/2382577.2382581.

[7] Jonathan Cohen (2009): *Graph Twiddling in a MapReduce World*. *Computing in Science and Engineering* 11(4), pp. 29–41, doi:10.1109/MCSE.2009.120. Available at `http://ieeexplore.ieee.org/document/5076317/`.

[8] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy & Russell Sears (2010): *MapReduce Online*. In: *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, USENIX Association, Berkeley, CA, USA, pp. 21–21. Available at `http://dl.acm.org/citation.cfm?id=1855711.1855732`.

[9] Camil Demetrescu (2010): *9th DIMACS Implementation Challenge*. `http://www.dis.uniroma1.it/challenge9/download.shtml`.

[10] Laboratori de Recerca i Desenvolupament (RDlab)-UPC (2016): *RdLab Manual*. `https://rdlab.cs.upc.edu/docu/html/manual_cluster/ClusterQuickstart_en.html`.

[11] Alan A.A. Donovan & Brian W. Kernighan (2015): *The Go Programming Language*, 1st edition. Addison-Wesley Professional.

[12] Go website. Google (2016): *Go version*. `https://blog.golang.org/go1.6`.

[13] Xiaocheng Hu, Miao Qiao & Yufei Tao (2015): *Join Dependency Testing, Loomis-Whitney Join, and Triangle Enumeration*. In: *Proceedings of the 34th ACM Symposium on Principles of Database Systems, PODS 2015, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pp. 291–301, doi:10.1145/2745754.2745768. Available at `http://dl.acm.org/citation.cfm?doid=2745754.2745768`.

[14] Xiaocheng Hu, Yufei Tao & Chin-Wan Chung (2014): *I/O-Efficient Algorithms on Triangle Listing and Counting*. *ACM Trans. Database Syst.* 39(4), pp. 27:1–27:30, doi:10.1145/2691190.2691193. Available at `http://dl.acm.org/citation.cfm?doid=2691190.2691193`.

[15] Konstantin Kutzkov & Rasmus Pagh (2014): *Triangle Counting in Dynamic Graph Streams*. In: *Algorithm Theory - SWAT 2014 - 14th Scandinavian Symposium and Workshops, Copenhagen, Denmark, July*

2-4, 2014. Proceedings, pp. 306–318, doi:10.1007/978-3-319-08404-6_27. Available at `http://dblp.uni-trier.de/rec/bib/conf/swat/KutzkovP14a`.

[16] Jure Leskovec (2016): *Social circles: Facebook*. `http://snap.stanford.edu/data/egonets-Facebook.html`.

[17] Jimmy Lin & Chris Dyer (2010): *Data-Intensive Text Processing with MapReduce*. Synthesis Lectures on Human Language Technologies, Morgan & Claypool Publishers, doi:10.2200/S00274ED1V01Y201006HLT007.

[18] Donald Miner & Adam Shook (2013): *MapReduce Design Patterns : building effective algorithms and analytics for Hadoop and other systems*. O'Reilly, Beijing, Kln, u.a. Available at `http://opac.inria.fr/record=b1134500`. DEBSZ.

[19] Rasmus Pagh & Charalampos E. Tsourakakis (2012): *Colorful triangle counting and a MapReduce implementation*. *Inf. Process. Lett.* 112(7), pp. 277–281, doi:10.1016/j.ipl.2011.12.007.

[20] Edelmira Pasarella, Maria-Esther Vidal & Cristina Zoltan (2016): *MapReduce vs. Pipelining Counting Triangles*. In: *Proceedings of the 10th Alberto Mendelzon International Workshop on Foundations of Data Management, Panama City, Panama, May 8-10, 2016*. Available at `http://ceur-ws.org/Vol-1644/paper33.pdf`.

[21] Mahmudur Rahman & Mohammad Al Hasan (2013): *Approximate triangle counting algorithms on multicores*. In: *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*, pp. 127–133, doi:10.1109/BigData.2013.6691744.

[22] Dryad Digital Repository (2016): *Dryad*. `http://datadryad.org/`.

[23] Siddharth Suri & Sergei Vassilvitskii (2011): *Counting Triangles and the Curse of the Last Reducer*. In: *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, ACM, New York, NY, USA, pp. 607–614, doi:10.1145/1963405.1963491.

[24] Leslie G. Valiant (1990): *A Bridging Model for Parallel Computation*. *Commun. ACM* 33(8), pp. 103–111, doi:10.1145/79173.79181.

[25] Tom White (2009): *Hadoop - The Definitive Guide: MapReduce for the Cloud*. O'Reilly. Available at `http://www.oreilly.de/catalog/9780596521974/index.html`.